JICSCI803
Algorithms and Data Structures 2019

Highlights of Lecture 10

Divide-and-Conquer Algorithms
Complexity =>have O(log n) factor

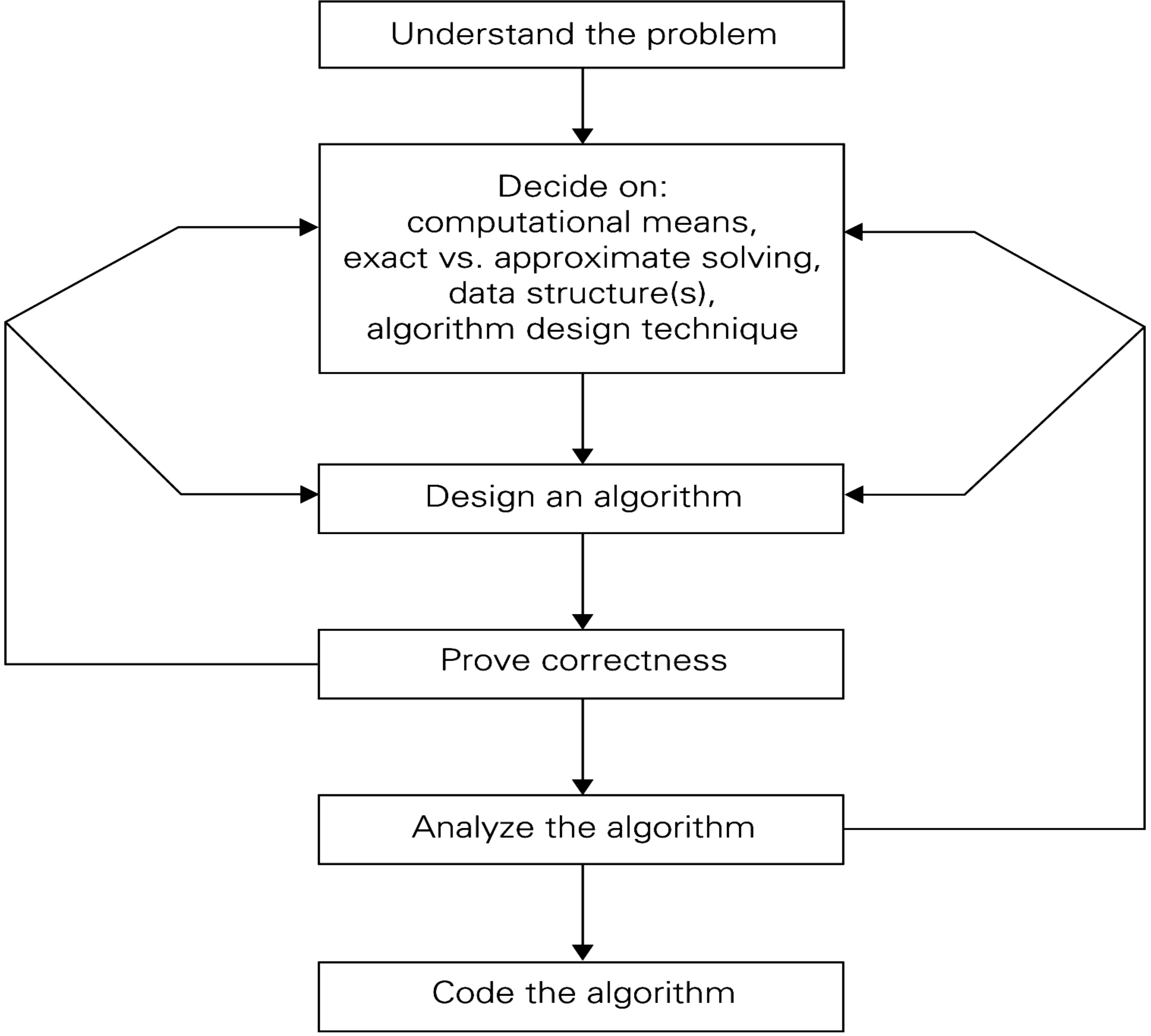
Examples: Mergesort; Binary Search; Integer Multiplication; Matrix Multiplication; FFT; Exponent Value; Polynomial
Point-Value.

Questions

Do you know how to execute a multiplication?

Do you know how to execute a multiplication in a fast way?

Algorithm Design and Analysis Process


Understand the problem
Decide on: computational means, exact vs. approximate solving, data structure(s), algorithm design technique
Design an algorithm
Prove correctness
Analyze the algorithm
Code the algorithm

Algorithm Analysis Framework

Measuring an input’s size
Measuring running time
Orders of growth (of the algorithm’s efficiency function)
Worst-base, best-case and average-case efficiency

Divide-and-Conquer Paradigm

Divide-and-Conquer Paradigm

In computer science, Divide and conquer (D&C) is an important algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Divide-and-Conquer Algorithms

This divide and conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. the Karatsuba algorithm), matrix multiplication, finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs), finding the value of exponential expressions, finding the point value of polynomials.

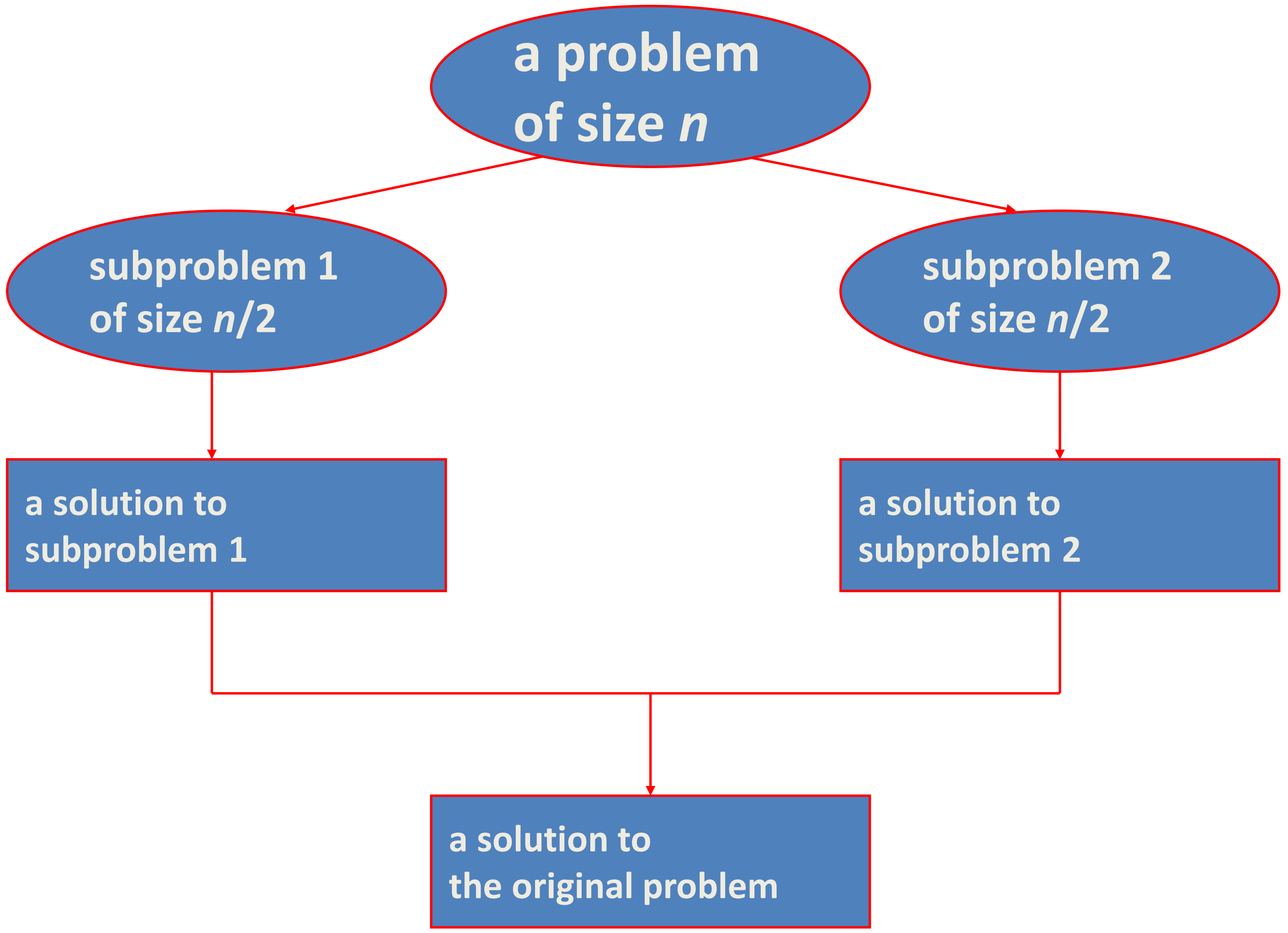
Divide-and-Conquer Algorithms

– In contrast to the greedy algorithms, the idea behind divide-and-conquer algorithms is to break the problem up into smaller sub problems which can each either be easily solved or further subdivided.

– Once the sub problems have been solved, the partial solutions are recombined to arrive at the solution of the original problem.

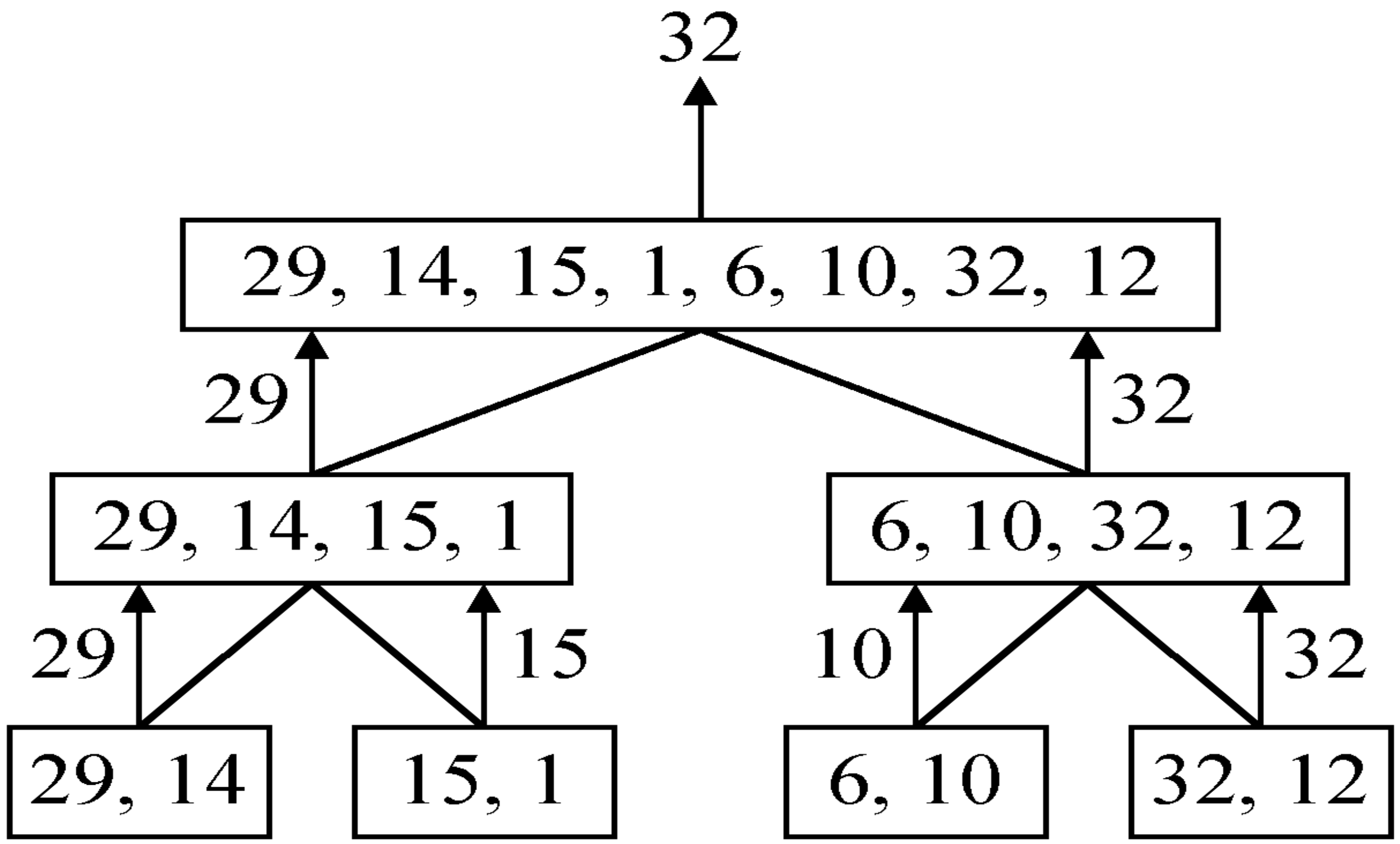
– We have already seen at least one example of a divide-and-conquer algorithm: quicksort.

Divide-and-Conquer Algorithms


a problem of size n
subproblem 1 of size $n/2$
subproblem 2 of size $n/2$
a solution to subproblem 1
a solution to subproblem 2
a solution to the original problem

A simple example

finding the maximum of a set S of n numbers



Time complexity of finding max

Time complexity:

T(n): # of comparisons

$$T(n)=\begin{cases} 2T(n/2)+1 & , n>2 \\ 1 & , n\le 2 \end{cases}$$

Calculation of T(n):

Assume $n = 2^k$,

$$\begin{aligned} T(n) &= 2T(n/2)+1 \\ &= 2(2T(n/4)+1)+1 \\ &= 4T(n/4)+2+1 \\ &\quad\vdots \\ &=2^{k-1}T(2)+2^{k-2}+\ldots+4+2+1 \\ &=2^{k-1}+2^{k-2}+\ldots+4+2+1 \\ &=2^k-1 = n-1 \end{aligned}$$

A 2-sub divide-and-conquer algorithm

Step 1: If the problem size is small, solve this problem directly; otherwise, split the original problem into 2 sub-problems with equal sizes.
Step 2: Recursively solve these 2 sub-problems by applying this algorithm.
Step 3: Merge the solutions of the 2 sub-problems into a solution of the original problem.

Time complexity of the general algorithm

Time complexity:

$$T(n) = \begin{cases} 2T(n/2)+S(n)+M(n) & , n \geq c \\ b & , n < c \end{cases}$$

where S(n) : time for splitting

M(n) : time for merging

b : a constant

c : a constant

General Divide-and-Conquer Algorithms

The general form of a D&C algorithm is as follows

```
Function D&C(x)
if x is easy enough to solve then
        return adhoc (x)
decompose x into subinstances  $x_1$ , $x_2$, $x_3$ , ... , $x_m$
for i = 1 to m do
    $y_i$ = D&C($x_i$)
recombine the $y_i$ (i = 1 to m ) to get y , the solution
for x
return y
```

Mergesort and Binary Search

D&C Algorithms-Mergesort

Split array A[1..*n*] in two and make copies of each half in arrays B[1.. *n*/2] and C[1.. *n*/2]

Sort arrays B and C

Merge sorted arrays B and C into array A as follows:

- Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
- Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

D&C Algorithms-Binary Search

Example 1: Binary Search

- Let T[1..n] be an array sorted in non-decreasing order; i.e. T[i] ≤ T[j] for all 1 ≤ i ≤ j ≤ n.
- We wish to find the position in T of a value x or, if x is not in T, the position where it should be inserted.

- Let us consider some instances:

Binary Search

– Consider the array *T* =

1	2	3	4	5	6	7	8	9	10	11
-5	-2	0	3	8	8	9	12	12	26	31

– search(-10) = 1 小于最小, 返回1

– search(-5) = 1

– search(0) = 3

– search(8) = 5

– search(13) = 10 不存在, 返回最后一个小于元素位置 + 1

– search(50) = 12 不存在, 返回最后一个小于元素位置 + 1

Binary Search

- At each level of the D&C algorithm we examine the value at the midpoint of the sub array.
- If this value is greater than or equal to the search value, we repeat the procedure with the left half of the sub array.
- If, on the other hand, the value is less than the search value we repeat the procedure with the right half.
- To avoid repeated tests in each recursive call it is better to determine if the answer is n + 1 at the start.

Binary Search

```
function binsearch(x, T[1..n])

if n = 0 or x > T[n] then return n + 1
else if x < T[1] return 1
else return binsrch(x, T(1..n])

function binsrch(x, T[i..j])
    if i = j then return i
    k = (i + j) / 2
    if x ≤ T[k] then
        return binsrch(x, T[i..k])
    else
        return binsrch(x, T[k + 1..j])
```

Binary Search (Program)

we can code an iterative version of binary search

```
function biniter(x, T[1..n])
  if n = 0 or x > T[n] then return n + 1

            i = 1; j = n
            while i < j do
               k = (i + j) / 2
               if x ≤ T[k] then
                   j = k
               else
                   i = k + 1
            return i
```

Finding the Sth Smallest Element

median

The **median** is the value separating the higher half of a data sample, a population, or a probability distribution, from the lower half. In simple terms, it may be thought of as the "middle" value of a data set. For example, in the data set {1, 3, 3, 6, 7, 8, 9}, the median is 6, the fourth number in the sample. The median is a commonly used measure of the properties of a data set in statistics and probability theory.

The basic advantage of the median in describing data compared to the mean (often simply described as the "average") is that it is not skewed so much by extremely large or small values, and so it may give a better idea of a 'typical' value. For example, in understanding statistics like household income or assets which vary greatly, a mean may be skewed by a small number of extremely high or low values. Median income, for example, may be a better way to suggest what a 'typical' income is.

D&C Algorithms-Finding the Median

Example 2: Finding the Median

- Let T[1 ..n] be an unsorted array.
- We wish to find the median value of the array T， the $(n/2)^{th}$ smallest element
- Clearly, we could sort T and select T[n/2]
- This approach would be Θ(n log n)
- Can we do better?
- Let us consider a more general problem
- Given an integer s, find the s^{th} *smallest element in T*

Median is the (n/2)th smallest element

- Finding the s^{th} Smallest Element
  - Consider a pivot function, similar to the quicksort pivot, which partitions T into three sub arrays.

```
function pivot3(T[i..j], p, q, r)
```

- Elements in T[i..q] are all < p
- Elements in T[q + 1..r – 1] are all = p
- Elements in T[r..j] are all > p

- Finding the s^{th} Smallest Element
  - If $s \leq q$ we recursively divide the first part of T
  - If $q < s < r$ the s^{th} element of T = T[s]
  - If $s \geq r$ we recursively divide the last part of T

  - Does this give a better than $\Theta(n \log n)$ algorithm?
  - Not unless we can use a special pivot
  - The median.

- Finding the *s*th Smallest Element
  - Consider the case where s = 4 and T is the array

3	1	4	1	5	9	2	6	5	3	5	8	9

  - Pivot this array around its median, p = 5

3	1	4	1	2	3	5	5	5	9	6	8	9

  - Pivot the left part around its median, p = 2

1	1	2	3	4	3	●	●	●	●	●	●	●

  - Pivot the right part around its median, p = 3

●	●	●	3	3	4	●	●	●	●	●	●	●

- Finding the s^{th} Smallest Element

```
function select(s, T[1..n])
    i = 1, j = n
    repeat
       p = median(T[i..j])
       pivot3(T[i..j], p, q, r)
       if s ≤ q then
           j = q
       else if s > r then
           i = r
       else
           return p
```

- Finding the s^{th} Smallest Element
  - Can we modify the algorithm so that we do not have to find the median?
  - Yes, the algorithm works with any pivot value
  - It is only the efficiency not the correctness which is affected
  - The worst case behavior if we were to use T[i] as the pivot is $\Theta(n^2)$ – like quicksort

Finding the s^{th} Smallest Element

– Perhaps we can find an approximation for the median

```
function pseudomed(T[1..n])
    if n ≤ 5 then
        return median5(T)
    z = n/5
    array Z[1..z]
    for i = 1 to z do
        Z[i] = median5(T[5i – 4..5i])
    return select(z/2,Z[1..z])
```

– function **`median5(X[1..5])`** returns the exact median of the array X

Finding the s^{th} Smallest Element

- How far from the true median of T can the result of pseudomed be?
- It can be shown that the rank of the result is roughly between 3n / 10 and 7n / 10
- How efficient is the select algorithm if we replace p = median(T[i..j]) with p = pseudomed(T[i..j])

- Finding the *s*th Smallest Element

– **Theorem:** the selection algorithm using the pseudomedian finds the *s*th smallest element from among *n* elements in Θ(n).

– **Proof:** is by induction and is too messy to consider here

Integer Multiplication

Integer Addition

Addition. Given two n-bit integers a and b, compute $a + b$.
In Naïve way, we need $\Theta(n)$ bit operations.

```
1   1   1   1   1   1   0   1
    1   1   0   1   0   1   0   1
+   0   1   1   1   1   1   0   1
---------------------------------
1   0   1   0   1   0   0   1   0
```

Remark. Naïve addition algorithm is optimal.

Integer Multiplication

Multiplication. Given two n-bit integers a and b, compute $a \times b$.
In Naïve way, we needs $\Theta(n^2)$ bit operations.

```
                1 1 0 1 0 1 0 1
              × 0 1 1 1 1 1 0 1
              -----------------
                1 1 0 1 0 1 0 1
              0 0 0 0 0 0 0 0 0
            1 1 0 1 0 1 0 1 0
          1 1 0 1 0 1 0 1 0
        1 1 0 1 0 1 0 1 0
      1 1 0 1 0 1 0 1 0
    1 1 0 1 0 1 0 1 0
  0 0 0 0 0 0 0 0 0
-------------------------------
0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 1
```

Divide-and-Conquer Multiplication

To multiply two *n*-bit integers *a* and *b*:

Multiply four ½*n*-bit integers, recursively.

Add and shift to obtain result.

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= \left(2^{n/2} \cdot a_1 + a_0\right)\left(2^{n/2} \cdot b_1 + b_0\right) = 2^n \cdot a_1 b_1 + 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0
\end{aligned}
$$

Ex.

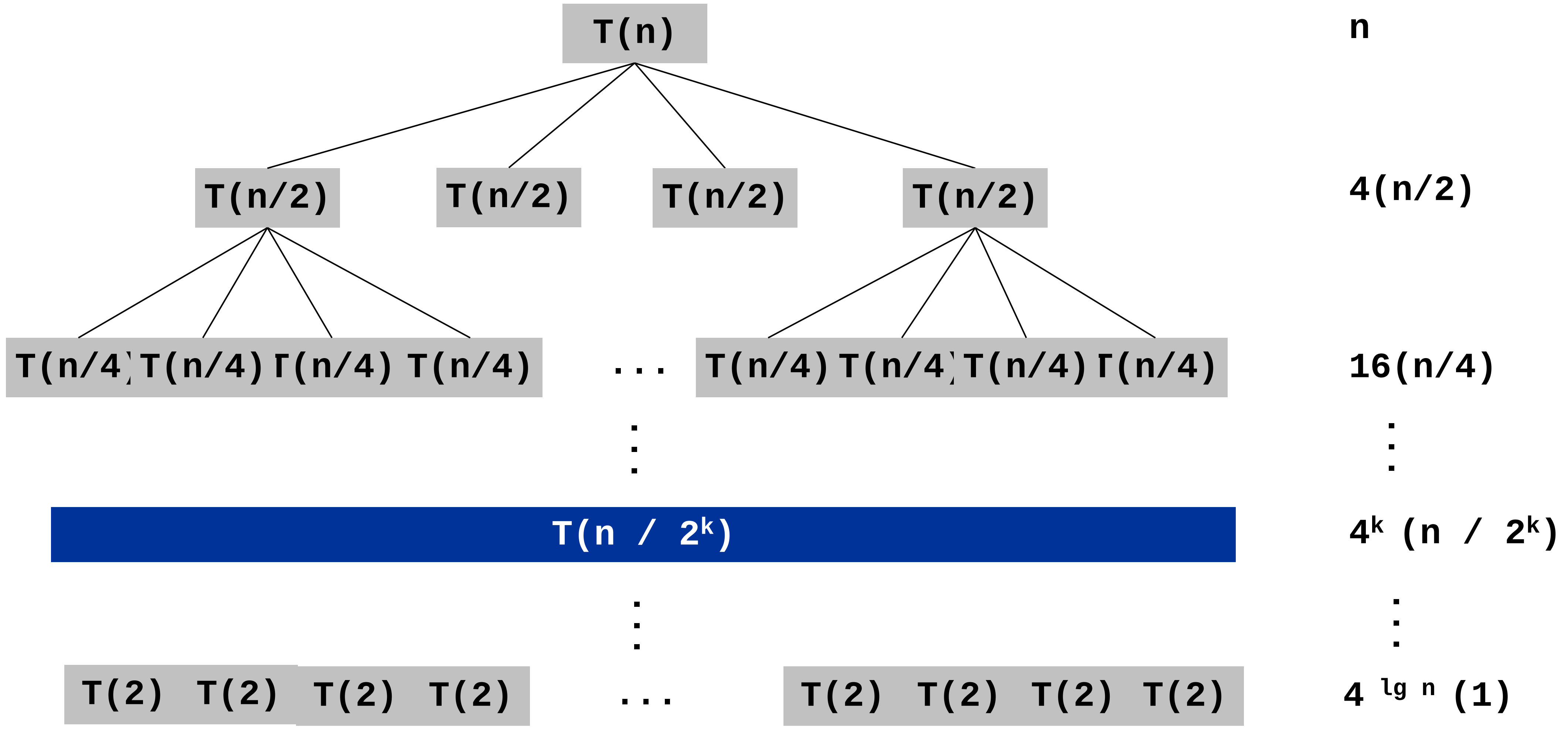
$$a = \underbrace{\mathtt{1000}}_{a_1}\underbrace{\mathtt{1101}}_{a_0} \qquad b = \underbrace{\mathtt{1110}}_{b_1}\underbrace{\mathtt{0001}}_{b_0}$$

$$T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow T(n) = \Theta(n^2)$$

Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 4T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\lg n} n\,2^k = n\left(\frac{2^{1+\lg n}-1}{2-1}\right) = 2n^2 - n$$



T(n)
n
T(n/2) T(n/2) T(n/2) T(n/2)
4(n/2)
T(n/4) T(n/4) T(n/4) T(n/4) . . . T(n/4) T(n/4) T(n/4) T(n/4)
16(n/4)
T(n / 2^k)
4^k (n / 2^k)
T(2) T(2) T(2) T(2) . . . T(2) T(2) T(2) T(2)
4 lg n (1)

Karatsuba Multiplication

To multiply two *n*-bit integers *a* and *b*:

- Add two ½*n* bit integers.
- Multiply three ½*n*-bit integers, recursively.
- Add, subtract, and shift to obtain result.

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^{n} \cdot a_1 b_1 + 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0 \\
&= 2^{n} \cdot \underset{\textcircled{1}}{a_1 b_1} + 2^{n/2} \cdot \left(\underset{\textcircled{2}}{(a_1 + a_0)(b_1 + b_0)} - \underset{\textcircled{1}}{a_1 b_1} - \underset{\textcircled{3}}{a_0 b_0}\right) + \underset{\textcircled{3}}{a_0 b_0}
\end{aligned}
$$

To multiply two *n*-bit integers *a* and *b*:

- Add two ½*n* bit integers.
- Multiply three ½*n*-bit integers, recursively.
- Add, subtract, and shift to obtain result.

$$
\begin{aligned}
a &= 2^{n/2}\cdot a_1 + a_0 \\
b &= 2^{n/2}\cdot b_1 + b_0 \\
ab &= 2^{n}\cdot a_1 b_1 + 2^{n/2}\cdot\left(a_1 b_0 + a_0 b_1\right) + a_0 b_0 \\
&= 2^{n}\cdot \underbrace{a_1 b_1}_{1} + 2^{n/2}\cdot\left(\underbrace{(a_1+a_0)(b_1+b_0)}_{2} - \underbrace{a_1 b_1}_{1} - \underbrace{a_0 b_0}_{3}\right) + \underbrace{a_0 b_0}_{3}
\end{aligned}
$$

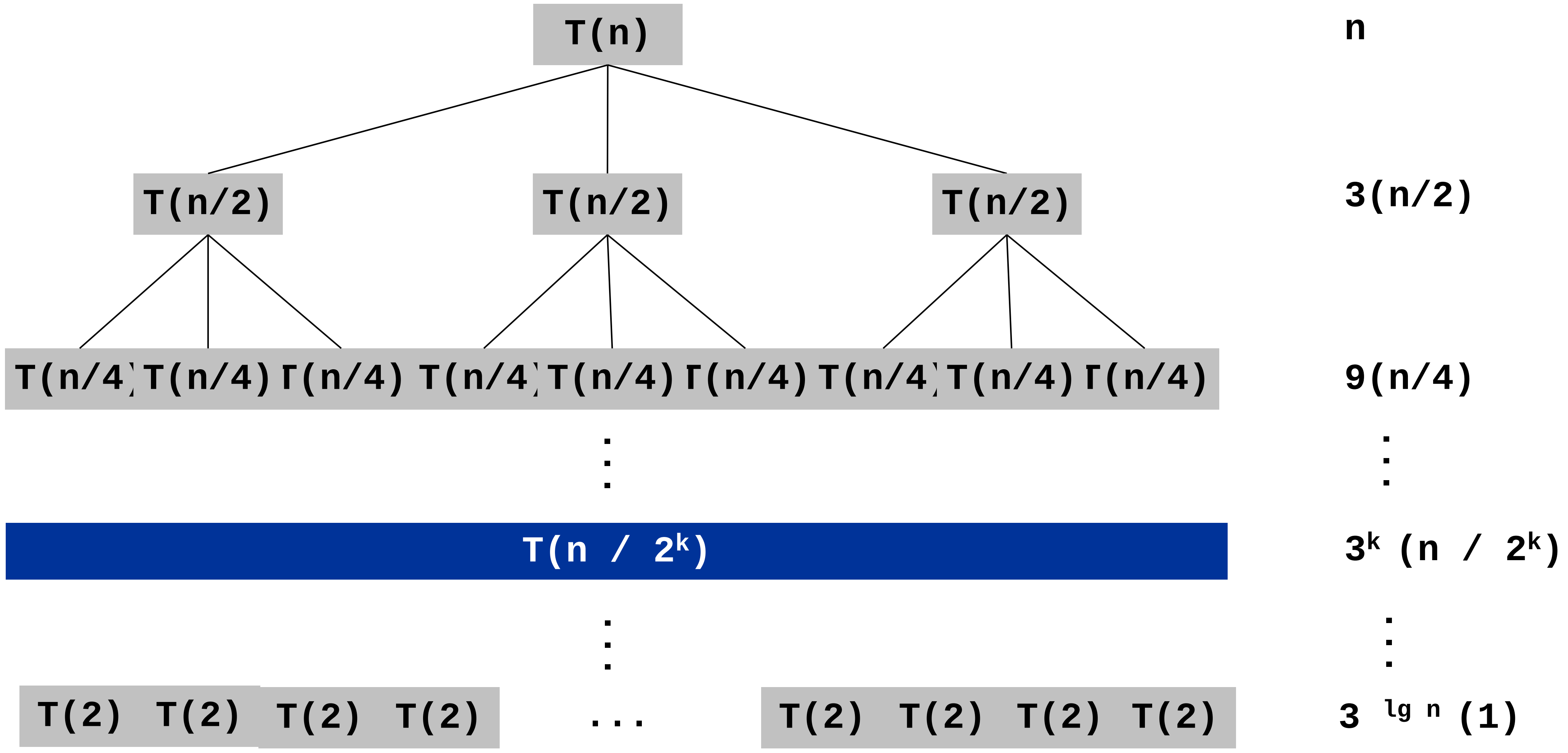
Theorem. [Karatsuba-Ofman 1962] Can multiply two *n*-bit integers in $O(n^{1.585})$ bit operations.

$$
T(n) \le \underbrace{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1+\lceil n/2 \rceil)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}} \Rightarrow T(n) = O(n^{\lg 3}) = O(n^{1.585})
$$

Karatsuba: Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\lg n} n \left(\tfrac{3}{2}\right)^k = n\left(\frac{\left(\tfrac{3}{2}\right)^{1+\lg n} - 1}{\tfrac{3}{2} - 1}\right) = 3n^{\lg 3} - 2n$$



T(n)
n
T(n/2)
T(n/2)
T(n/2)
3(n/2)
T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) T(n/4)
9(n/4)
T(n / 2^k)
3^k (n / 2^k)
T(2) T(2) T(2) T(2) . . . T(2) T(2) T(2) T(2)
3 ^lg n (1)

Karatsuba Algorithm

The Karatsuba algorithm is a fast multiplication algorithm. It was discovered by Anatoly Karatsuba in 1960 and published in 1962. It reduces the multiplication of two n-digit numbers to at most about $n^{1.585}$ single-digit multiplications in general. It is therefore faster than the classical algorithm, which requires n^2 single-digit products. For example, the Karatsuba algorithm requires $3^{10} = 59{,}049$ single-digit multiplications to multiply two 1024-digit numbers ($n = 1024 = 2^{10}$), whereas the classical algorithm requires $(2^{10})^2 = 1{,}048{,}576$.

Karatsuba Algorithm

Two n-digit numbers needs $n^{1.585}$ single-digit multiplications in general.

For example, two 1024-digit numbers multiplications.

In the classical algorithm requires $(2^{10})^2 = 1{,}048{,}576$.

In the Karatsuba algorithm requires $n^{1.585} = (2^{10})^{1.585} = (2^{1.585})^{10} \approx 3^{10} = 59{,}049$.

Notice that $(n = 1024 = 2^{10})$.

Karatsuba Algorithm (Decimal Number)

Using the classical pen and paper algorithm two n digit integers can be multiplied in $O(n^2)$ operations. Karatsuba came up with a faster algorithm.

Let A and B be two integers with

$A = A_1 10^k + A_0,\ A_0 < 10^k$

$B = B_1 10^k + B_0,\ B_0 < 10^k$

$C = A*B = (A_1 10^k + A_0)(B_1 10^k + B_0)$

$\quad = A_1 B_1 10^{2k} + (A_1 B_0 + A_0 B_1) 10^k + A_0 B_0$

Instead this can be computed with 3 multiplications

$T_0 = A_0 B_0$

$T_1 = (A_1 + A_0)(B_1 + B_0)$

$T_2 = A_1 B_1$

$C = T_2 10^{2k} + (T_1 - T_0 - T_2) 10^k + T_0$

Worked Example

- Compute 1234 * 4321.
- Subproblems:
  - $a_1 = 12 * 43$
  - $d_1 = 34 * 21$
  - $e_1 = (12 + 34) * (43 + 21) - a_1 - d_1$
    $= 46 * 64 - a_1 - d_1$
  - Need to recurse…

Worked Example

- First subproblem:

  $a_1 = 12 * 43$

- Subproblems:
  - $a_2 = 1 * 4 = 4$
  - $d_2 = 2 * 3 = 6$
  - $e_2 = (1+2)(4+3) – a2 – d2$
    $= 11$
- Answer: $4 * 10^2 + 11 * 10 + 6 = 516$

Worked Example

- Second subproblem

$$d_1 = 34 * 21$$

- Subproblems:
  - $a_2 = 3 * 2 = 6$
  - $d_2 = 4 * 1 = 4$
  - $e_2 = (3+4)(2+1) - a2 - d2$
    $= 11$
- Answer: $6 * 10^2 + 11 * 10 + 4 = 714$

Worked Example

- Third subproblem:

  $e_1 = 46 * 64 - a_1 - d_1$

- Subproblems:
  - $a_2 = 4 * 6 = 24$
  - $d_2 = 6 * 4 = 24$
  - $e_2 = (4+6)(6+4) - a2 - d2$

    $= 52$

- Answer: $e_1 = 24 * 10^2 + 52 * 10 + 24 - 714 - 516$

  $= 1714$

Worked Example

- Final Answer:

$1234 * 4321 = 516 * 10^4 + 1714 * 10^2 + 714$

$= 5{,}332{,}114$

Matrix Multiplication

Matrix Multiplication : example 4

General Matrix Multiplication

Brute-force algorithm

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

$$= \begin{pmatrix} a_{00} * b_{00} + a_{01} * b_{10} & a_{00} * b_{01} + a_{01} * b_{11} \\ a_{10} * b_{00} + a_{11} * b_{10} & a_{10} * b_{01} + a_{11} * b_{11} \end{pmatrix}$$

8 multiplications

4 additions

Efficiency class in general: Θ (n^3)

Strassen's Matrix Multiplication

Strassen's algorithm for two 2x2 matrices (1969):

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
$m_2 = (a_{10} + a_{11}) * b_{00}$
$m_3 = a_{00} * (b_{01} - b_{11})$
$m_4 = a_{11} * (b_{10} - b_{00})$
$m_5 = (a_{00} + a_{01}) * b_{11}$
$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

7 multiplications

18 additions

Strassen’s matrix multiplication

Strassen observed [1969] that the product of two matrices can be computed as follows:

$$\left(\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right) = \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right) * \left(\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right)$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

Submatrices:

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$

$M_2 = (A_{10} + A_{11}) * B_{00}$

$M_3 = A_{00} * (B_{01} - B_{11})$

$M_4 = A_{11} * (B_{10} - B_{00})$

$M_5 = (A_{00} + A_{01}) * B_{11}$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

Matrix multiplication

Let A, B and C be $n \times n$ matrices

$$C = AB$$

$$C(i, j) = \sum_{1 \le k \le n} A(i, k)B(k, j)$$

The straightforward method to perform a matrix multiplication requires $O(n^3)$ time.

Divide-and-conquer approach

$C = AB$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \begin{matrix} = \\ = \end{matrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$C_{11} = A_{11} B_{11} + A_{12} B_{21}$

$C_{12} = A_{11} B_{12} + A_{12} B_{22}$

$C_{21} = A_{21} B_{11} + A_{22} B_{21}$

$C_{22} = A_{21} B_{12} + A_{22} B_{22}$

Time complexity: (# of additions : n^2)

We get $T(n) = O(n^3)$

$$T(n) = \begin{cases} b & , n \le 2 \\ 8T(n/2)+cn^2 & , n > 2 \end{cases}$$

Strassen's matrix multiplicaiton

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = (A_{21} + A_{22})B_{11}$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = (A_{11} + A_{12})B_{22}$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$.

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

$$\boxed{\begin{aligned} C_{11} &= A_{11} B_{11} + A_{12} B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12} B_{22} \\ C_{21} &= A_{21} B_{11} + A_{22} B_{21} \\ C_{22} &= A_{21} B_{12} + A_{22} B_{22} \end{aligned}}$$

Time complexity

7 multiplications and 18 additions or subtractions

Time complexity:

$$T(n) = \begin{cases} b & , n \le 2 \\ 7T(n/2)+an^2 & , n > 2 \end{cases}$$

$$\begin{aligned}
T(n) &= an^2 + 7T(n/2) \\
&= an^2 + 7(a(\tfrac{n}{2})^2 + 7T(n/4) \\
&= an^2 + \tfrac{7}{4}an^2 + 7^2T(n/4) \\
&= \cdots \\
&\quad \vdots \\
&= an^2(1 + \tfrac{7}{4} + (\tfrac{7}{4})^2 + \cdots + (\tfrac{7}{4})^{k-1}) + 7^kT(1) \\
&\le cn^2(\tfrac{7}{4})^{\log_2 n} + 7^{\log_2 n}, \quad c \text{ is a constant} \\
&= cn^2(\tfrac{7}{4})^{\log_2 n} + n^{\log_2 7} = cn^{\log_2 4 - \log_2 7 + \log_2 4} + n^{\log_2 7} \\
&= \mathrm{O}(n^{\log_2 7}) \cong \mathrm{O}(n^{2.81})
\end{aligned}$$

Fast Fourier transform (FFT)

Fast Fourier transform (FFT) : example 5

Fourier transform $$b(f) = \int_{-\infty}^{\infty} a(t) e^{i2\pi ft} dt, \text{ where } i = \sqrt{-1}$$

Inverse Fourier transform $$a(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} b(f) e^{-i2\pi ft} dt$$

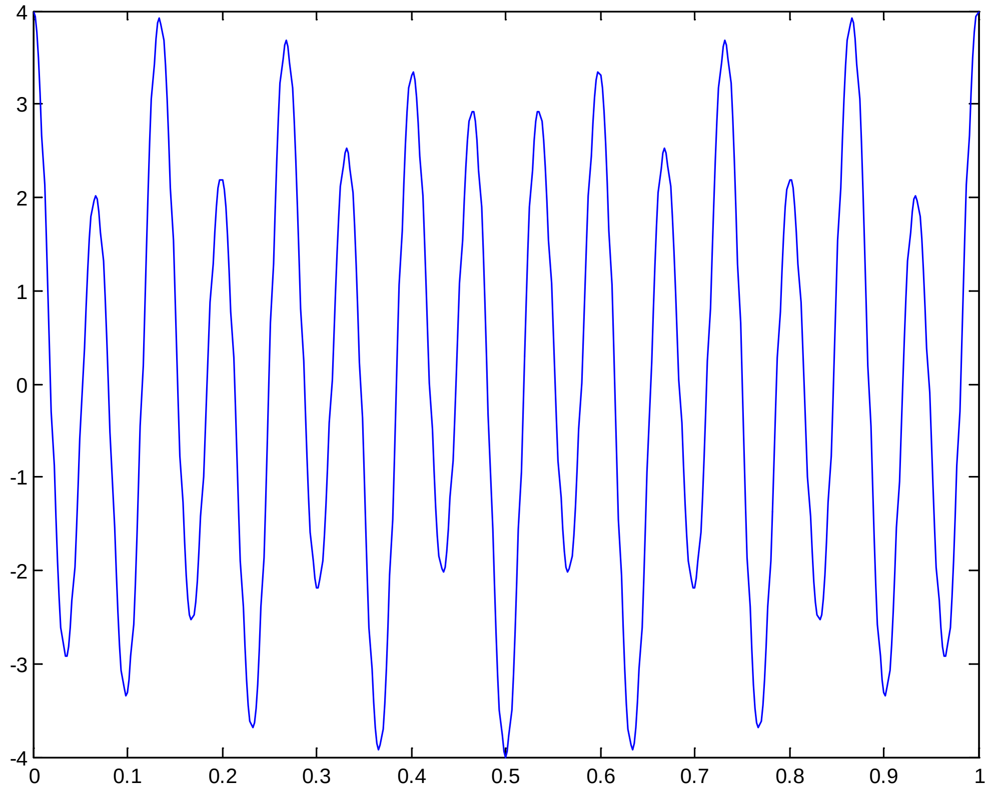
Discrete Fourier transform(DFT)

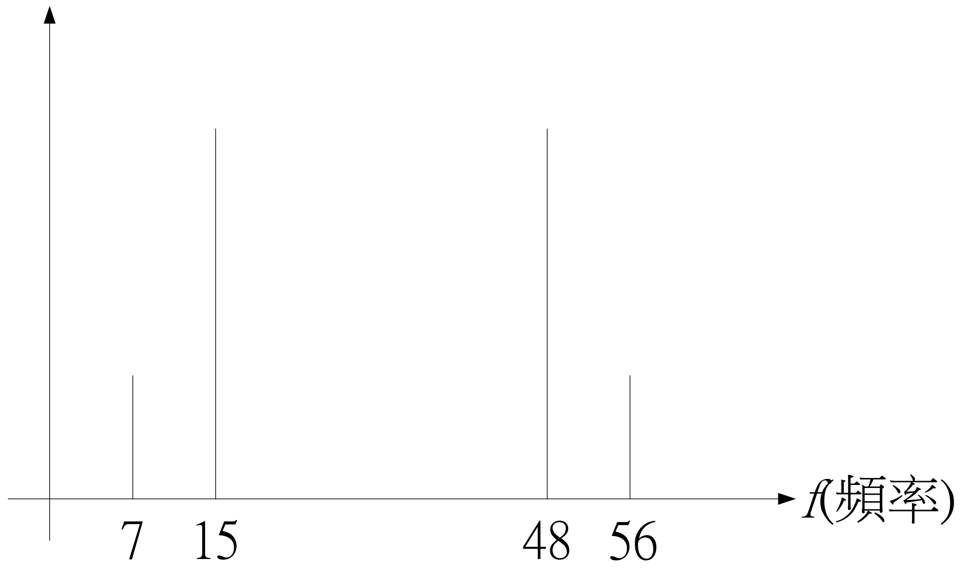
Given $a_0, a_1, \ldots, a_{n-1}$, compute

$$\begin{aligned} b_j &= \sum_{k=0}^{n-1} a_k e^{i2\pi jk/n}, 0 \le j \le n-1 \\ &= \sum_{k=0}^{n-1} a_k \omega^{kj}, \text{ where } \omega = e^{i2\pi/n} \end{aligned}$$

DFT and waveform(1)

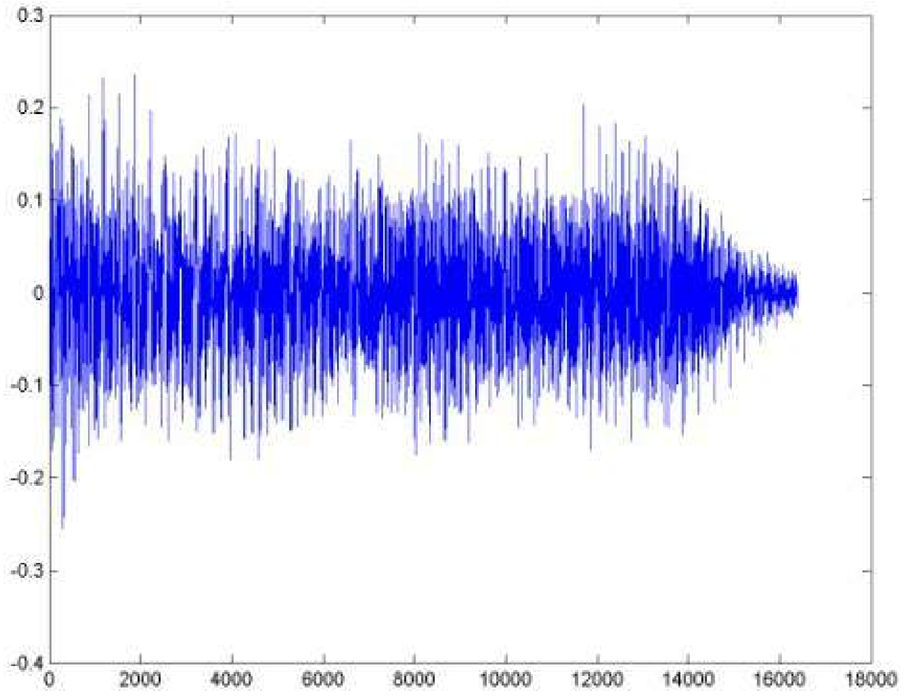
Any periodic waveform can be decomposed into the linear sum of sinusoid functions (sine or cosine).



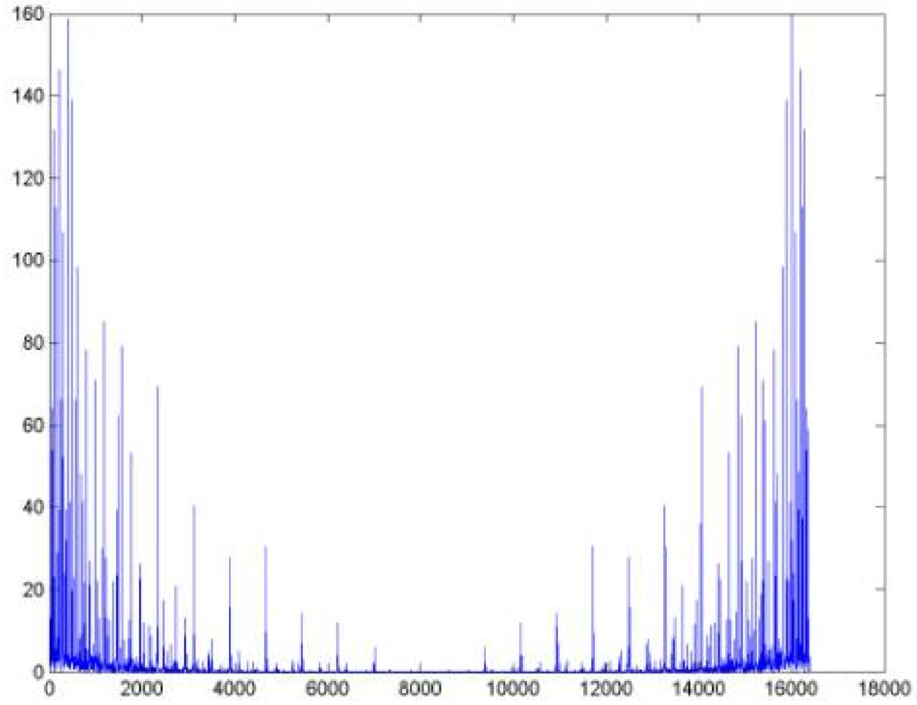


$$f(t) = \cos(2\pi(7)t) + 3\cos(2\pi(15)t) + 3\cos(2\pi(48)t) + \cos(2\pi(56)t)$$

DFT and waveform (2)



The waveform of a music signal of 1 second

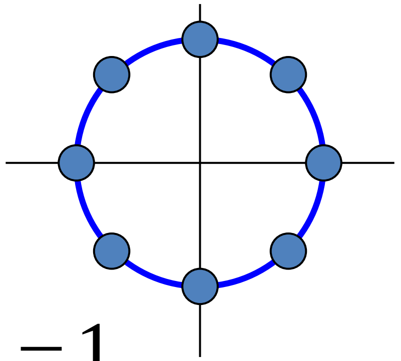


The frequency spectrum of the music signal with DFT

FFT algorithm

Inverse DFT:

$$a_k = \frac{1}{n}\sum_{j=0}^{n-1} b_j \omega^{-jk}, 0 \le k \le n-1$$



$$e^{i\theta} = \cos\theta + i\sin\theta$$

$$\omega^n = (e^{i2\pi/n})^n = e^{i2\pi} = \cos 2\pi + i\sin 2\pi = 1$$

$$\omega^{n/2} = (e^{i2\pi/n})^{n/2} = e^{i\pi} = \cos\pi + i\sin\pi = -1$$

DFT can be computed in O(n^2) time by a straightforward method.
DFT can be solved by the divide-and-conquer strategy (FFT) in O($n$log$n$) time.

FFT algorithm when n=4

$n=4$, $w=e^{i2\pi/4}$, $w^4=1$, $w^2=-1$

$b_0=a_0+a_1+a_2+a_3$

$b_1=a_0+a_1w+a_2w^2+a_3w^3$

$b_2=a_0+a_1w^2+a_2w^4+a_3w^6$

$b_3=a_0+a_1w^3+a_2w^6+a_3w^9$

another form:

$b_0 \ =(a_0+a_2)+(a_1+a_3)$

$b_2 \ =(a_0+a_2w^4)+(a_1w^2+a_3w^6) =(a_0+a_2)-(a_1+a_3)$

When we calculate b_0, we shall calculate (a_0+a_2) and (a_1+a_3). Later, b_2 can be easily calculated.

Similarly,

$b_1 \ =(a_0+ a_2w^2)+(a_1w+a_3w^3) =(a_0-a_2)+w(a_1-a_3)$

$b_3 \ =(a_0+a_2w^6)+(a_1w^3+a_3w^9) =(a_0-a_2)-w(a_1-a_3)$.

FFT algorithm when n=8

n=8, $w=e^{i2\pi/8}$, $w^8=1$, $w^4=-1$

$b_0=a_0+a_1+a_2+a_3+a_4+a_5+a_6+a_7$

$b_1=a_0+a_1w+a_2w^2+a_3w^3+a_4w^4+a_5w^5+a_6w^6+a_7w^7$

$b_2=a_0+a_1w^2+a_2w^4+a_3w^6+a_4w^8+a_5w^{10}+a_6w^{12}+a_7w^{14}$

$b_3=a_0+a_1w^3+a_2w^6+a_3w^9+a_4w^{12}+a_5w^{15}+a_6w^{18}+a_7w^{21}$

$b_4=a_0+a_1w^4+a_2w^8+a_3w^{12}+a_4w^{16}+a_5w^{20}+a_6w^{24}+a_7w^{28}$

$b_5=a_0+a_1w^5+a_2w^{10}+a_3w^{15}+a_4w^{20}+a_5w^{25}+a_6w^{30}+a_7w^{35}$

$b_6=a_0+a_1w^6+a_2w^{12}+a_3w^{18}+a_4w^{24}+a_5w^{30}+a_6w^{36}+a_7w^{42}$

$b_7=a_0+a_1w^7+a_2w^{14}+a_3w^{21}+a_4w^{28}+a_5w^{35}+a_6w^{42}+a_7w^{49}$

After reordering, we have

$b_0=(a_0+a_2+a_4+a_6)+(a_1+a_3+a_5+a_7)$

$b_1=(a_0+a_2w^2+a_4w^4+a_6w^6)+ w(a_1+a_3w^2+a_5w^4+a_7w^6)$

$b_2=(a_0+a_2w^4+a_4w^8+a_6w^{12})+ w^2(a_1+a_3w^4+a_5w^8+a_7w^{12})$

$b_3=(a_0+a_2w^6+a_4w^{12}+a_6w^{18})+ w^3(a_1+a_3w^6+a_5w^{12}+a_7w^{18})$

$b_4=(a_0+a_2+a_4+a_6)-(a_1+a_3+a_5+a_7)$

$b_5=(a_0+a_2w^2+a_4w^4+a_6w^6)-w(a_1+a_3w^2+a_5w^4+a_7w^6)$

$b_6=(a_0+a_2w^4+a_4w^8+a_6w^{12})-w^2(a_1+a_3w^4+a_5w^8+a_7w^{12})$

$b_7=(a_0+a_2w^6+a_4w^{12}+a_6w^{18})-w^3(a_1+a_3w^6+a_5w^{12}+a_7w^{18})$

Rewrite as

$b_0=c_0+d_0$	$b_4=c_0-d_0=c_0+w^4d_0$
$b_1=c_1+wd_1$	$b_5=c_1-wd_1=c_1+w^5d_1$
$b_2=c_2+w^2d_2$	$b_6=c_2-w^2d_2=c_2+w^6d_2$
$b_3=c_3+w^3d_3$	$b_7=c_3-w^3d_3=c_3+w^7d_3$

$$c_0=a_0+a_2+a_4+a_6$$
$$c_1=a_0+a_2w^2+a_4w^4+a_6w^6$$
$$c_2=a_0+a_2w^4+a_4w^8+a_6w^{12}$$
$$c_3=a_0+a_2w^6+a_4w^{12}+a_6w^{18}$$

Let $x=w^2=e^{i2\pi/4}$

$$c_0=a_0+a_2+a_4+a_6$$
$$c_1=a_0+a_2x+a_4x^2+a_6x^3$$
$$c_2=a_0+a_2x^2+a_4x^4+a_6x^6$$
$$c_3=a_0+a_2x^3+a_4x^6+a_6x^9$$

Thus, $\{c_0,c_1,c_2,c_3\}$ is FFT of $\{a_0,a_2,a_4,a_6\}$.

Similarly, $\{d_0,d_1,d_2,d_3\}$ is FFT of $\{a_1,a_3,a_5,a_7\}$.

General FFT

In general, let $w=e^{i2\pi/n}$ (assume n is even.)

$$w^n=1,\ w^{n/2}=-1$$

$$b_j = a_0+a_1w^j+a_2w^{2j}+\ldots+a_{n-1}w^{(n-1)j},$$
$$= \{a_0+a_2w^{2j}+a_4w^{4j}+\ldots+a_{n-2}w^{(n-2)j}\}+$$
$$w^j\{a_1+a_3w^{2j}+a_5w^{4j}+\ldots+a_{n-1}w^{(n-2)j}\}$$
$$=c_j+w^jd_j$$
$$b_{j+n/2}=a_0+a_1w^{j+\mathrm{n}/2}+a_2w^{2j+\mathrm{n}}+a_3w^{3j+3n/2}+\ldots$$
$$+a_{n-1}w^{(n-1)j+n(n-1)/2}$$
$$=a_0-a_1w^j+a_2w^{2j}-a_3w^{3j}+\ldots+a_{n-2}w^{(n-2)j}-a_{n-1}w^{(n-1)j}$$
$$=c_j-w^jd_j$$
$$=c_j+w^{j+n/2}d_j$$

Divide-and-conquer (FFT)

Input: $a_0, a_1, \ldots, a_{n-1}$, $n = 2^k$
Output: b_j, $j=0, 1, 2, \ldots, n-1$
where $b_j = \sum_{0 \le k \le n-1} a_k w^{kj}$, where $w = e^{i2\pi/n}$

Step 1: If n=2, compute
$b_0 = a_0 + a_1$,
$b_1 = a_0 - a_1$, and return.
Step 2: Recursively find the Fourier transform of $\{a_0, a_2, a_4, \ldots, a_{n-2}\}$ and $\{a_1, a_3, a_5, \ldots, a_{n-1}\}$, whose results are denoted as $\{c_0, c_1, c_2, \ldots, c_{n/2-1}\}$ and $\{d_0, d_1, d_2, \ldots, d_{n/2-1}\}$.

Step 3: Compute b_j:

$$b_j = c_j + w^j d_j \quad \text{for} \quad 0 \le j \le n/2 - 1$$

$$b_{j+n/2} = c_j - w^j d_j \quad \text{for} \quad 0 \le j \le n/2 - 1.$$

Time complexity:

$$T(n) = 2T(n/2) + O(n)$$
$$= O(n \log n)$$

D&C Algorithms- a^n

Evaluating The value of a^n

D&C Algorithms-a^n

```
function exp1(a, n)
    x = 1
    for i = 1 to n do
        x = x * a
    return x
```

– This algorithm takes *n* multiplications
– Can we do better?

D&C Algorithms-a^n

– We note that:

$$a^n = \begin{cases} a & \text{if } n = 1 \\ (a^{n/2})^2 & \text{if } n \text{ is even} \\ a \times a^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

– Thus for example $a^{29} = a\ a^{28} = a\ (a^{14})^2$ …..

– Which involves 7 multiplications compared to 28

– Which leads to the following algorithm:

n

D&C Algorithms-a^n

```
Function expo2(a, n)
    if n = 1 then
        return a
    if n is even then
        return (expo2(a, n/2))²
    else
        return a * expo2(a, n-1)
```

– How efficient is this algorithm?

D&C Algorithms-a^n

– Let *N* (n) be the number of multiplications involved in calculating a^n with expo2

$$N(n) = \begin{cases} 0 & \text{if } n = 1 \\ N(n/2) + 1 & \text{if } n \text{ is even} \\ N(n-1) + 1 & \text{if } n \text{ is odd} \end{cases}$$

– This does not lead to an easy non-recursive evaluation of *N* (*n*)
– It can be shown that $N(n) \in \Theta(\log(n))$

D&C Algorithms-a^n

– We can also produce an iterative equivalent of this algorithm

```
function expo3(a, n)
      i = n
      result = 1
      x = a
      while i > 0 do
            if i is odd then
                  result = result * x
            x = x * x
            i = i/2
      return result
```

Worked Example

2^7

i = 7 ; result = 1; x= 2.

i = 7 is odd :   result = 2; x=4; i=3.

i = 3 is odd :   result = 8; x=16; i=1.

i = 1 is odd :   result = 128; x=256; i=0.   result = 128.

2^9

i = 9 ; result = 1; x= 2.

i = 9 is odd   :   result = 2; x=4; i=4.

i = 4 is even :   result = 2; x=16; i=2.

i = 2 is even :   result = 2; x=256; i=1.

i = 1 is odd   :   result = 2; x=512; i=0.   result = 512.

A Modest PhD Dissertation Title

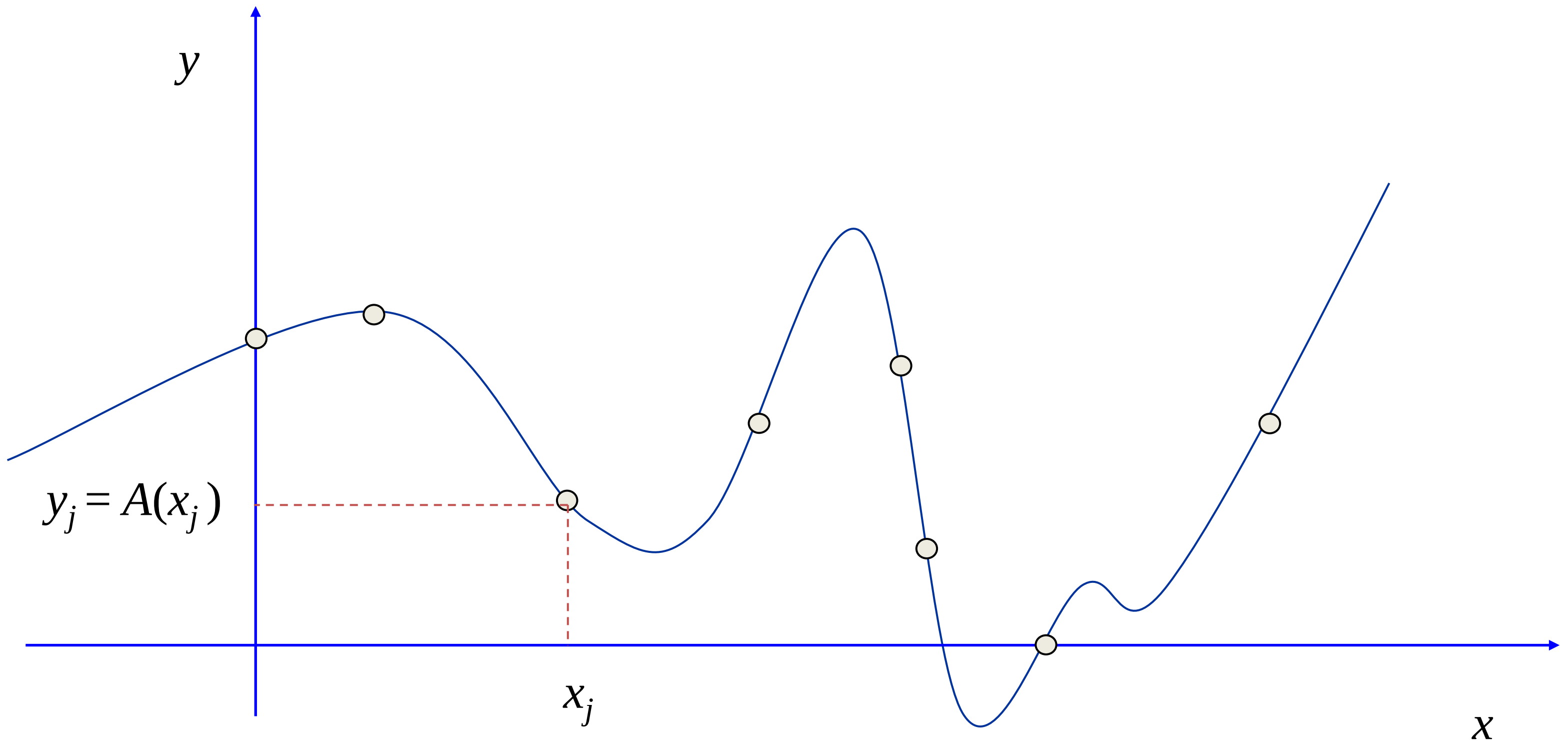
"New Proof of the Theorem That Every Algebraic Rational Integral Function In One Variable can be Resolved into Real Factors of the First or the Second Degree."

- PhD dissertation, 1799 the University of Helmstedt

Polynomials: Point-Value Representation

Fundamental theorem of algebra. [Gauss, PhD thesis] A degree *n* polynomial with complex coefficients has exactly *n* complex roots.

Corollary. A degree *n*-1 polynomial *A*(*x*) is uniquely specified by its evaluation at *n* distinct values of *x*.



Polynomials: Point-Value Representation

Polynomial. [point-value representation]

$$A(x):\ (x_0,\ y_0),\ \ldots,\ (x_{n-1},\ y_{n-1})$$

$$B(x):\ (x_0,\ z_0),\ \ldots,\ (x_{n-1},\ z_{n-1})$$

Add. $O(n)$ arithmetic operations.

$$A(x)+B(x):\ (x_0,\ y_0+z_0),\ \ldots,\ (x_{n-1},\ y_{n-1}+z_{n-1})$$

Multiply (convolve). $O(n)$, but need $2n$-1 points.

$$A(x)\times B(x):\ (x_0,\ y_0\times z_0),\ \ldots,\ (x_{2n-1},\ y_{2n-1}\times z_{2n-1})$$

Evaluate. $O(n^2)$ using Lagrange's formula.

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j\neq k}(x-x_j)}{\prod_{j\neq k}(x_k-x_j)}$$

Converting Between Two Polynomial Representations

Tradeoff. Fast evaluation or fast multiplication. We want both!

representation	multiply	evaluate
coefficient	$O(n^2)$	$O(n)$
point-value	$O(n)$	$O(n^2)$

Goal. Efficient conversion between two representations $\Rightarrow$ all ops fast.

$$\begin{tikzcd}
\boxed{a_0, a_1, \ldots, a_{n-1}} \arrow[r, shift left] & \boxed{(x_0, y_0), \square, (x_{n-1}, y_{n-1})} \arrow[l, shift left]
\end{tikzcd}$$

coefficient representation

point-value representation

Converting Between Two Representations: Brute Force

Coefficient $\Rightarrow$ point-value. Given a polynomial $a_0 + a_1 x + ... + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, ..., x_{n-1}$.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \square \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \square & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \square & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \square & x_2^{n-1} \\ \square & \square & \square & \square & \square \\ 1 & x_{n-1} & x_{n-1}^2 & \square & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \square \\ a_{n-1} \end{bmatrix}$$

Running time. $O(n^2)$ for matrix-vector multiply (or n Horner's).

Converting Between Two Representations: Brute Force

Point-value $\Rightarrow$ coefficient. Given n distinct points $x_0, \ldots, x_{n-1}$ and values $y_0, \ldots, y_{n-1}$, find unique polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, that has given values at given points.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \square \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \square & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \square & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \square & x_2^{n-1} \\ \square & \square & \square & \square & \square \\ 1 & x_{n-1} & x_{n-1}^2 & \square & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \square \\ a_{n-1} \end{bmatrix}$$

Vandermonde matrix is invertible iff x_i distinct

Running time. $O(n^3)$ for Gaussian elimination.

or $O(n^{2.376})$ via fast matrix multiplication

Divide-and-Conquer

Decimation in frequency. Break up polynomial into low and high powers.

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7.$$

$$A_{low}(x) = a_0 + a_1x + a_2x^2 + a_3x^3.$$

$$A_{high}(x) = a_4 + a_5x + a_6x^2 + a_7x^3.$$

$$A(x) = A_{low}(x) + x^4 A_{high}(x).$$

Decimation in time. Break polynomial up into even and odd powers.

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7.$$

$$A_{even}(x) = a_0 + a_2x + a_4x^2 + a_6x^3.$$

$$A_{odd}(x) = a_1 + a_3x + a_5x^2 + a_7x^3.$$

$$A(x) = A_{even}(x^2) + x\,A_{odd}(x^2).$$

Coefficient to Point-Value Representation: Intuition

Coefficient $\Rightarrow$ point-value. Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$.

we get to choose which ones!

Divide. Break polynomial up into even and odd powers.

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7.$$
$$A_{even}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3.$$
$$A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3.$$
$$A(x) = A_{even}(x^2) + x\,A_{odd}(x^2).$$
$$A(-x) = A_{even}(x^2) - x\,A_{odd}(x^2).$$

Intuition. Choose two points to be ± 1.

$$A(1) = A_{even}(1) + 1\,A_{odd}(1).$$
$$A(-1) = A_{even}(1) - 1\,A_{odd}(1).$$

Can evaluate polynomial of degree $\leq n$ at 2 points by evaluating two polynomials of degree $\leq \frac{1}{2}n$ at 1 point.

Coefficient to Point-Value Representation: Intuition

Coefficient $\Rightarrow$ point-value. Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$.

we get to choose which ones!

Divide. Break polynomial up into even and odd powers.

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7.$$
$$A_{even}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3.$$
$$A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3.$$
$$A(x) = A_{even}(x^2) + x\, A_{odd}(x^2).$$
$$A(-x) = A_{even}(x^2) - x\, A_{odd}(x^2).$$

Intuition. Choose four complex points to be ± 1, $\pm i$.

$$A(1) = A_{even}(1) + 1\, A_{odd}(1).$$
$$A(-1) = A_{even}(1) - 1\, A_{odd}(1).$$
$$A(i) = A_{even}(-1) + i\, A_{odd}(-1).$$
$$A(-i) = A_{even}(-1) - i\, A_{odd}(-1).$$

Can evaluate polynomial of degree $\leq n$ at 2 points by evaluating two polynomials of degree $\leq \frac{1}{2}n$ at 1 point.

Discussions

1. What is D&C algorithm paradigm?
2. Which are the problems we have applied this paradigm?
3. What are the real cases we can apply this paradigm?

Homework

1. Implement **`exp1(a, n)`, `exp2(a, n)`, and `exp3(a, n)` and compare the running time for a = 7, n=1 to 10.**